The Singleton Pattern

Human Computer Interaction Research
University of Nevada, Reno

Singleton

- Behavioral Patterns
  - » strategy
  - » observer
  - » decorator
- Creational Patterns
  - » factory method
  - » abstract factory
  - » singleton

problem

"Some classes only need to be instantiated once otherwise you get incorrect behavior, overuse of resources and inconsistent results"

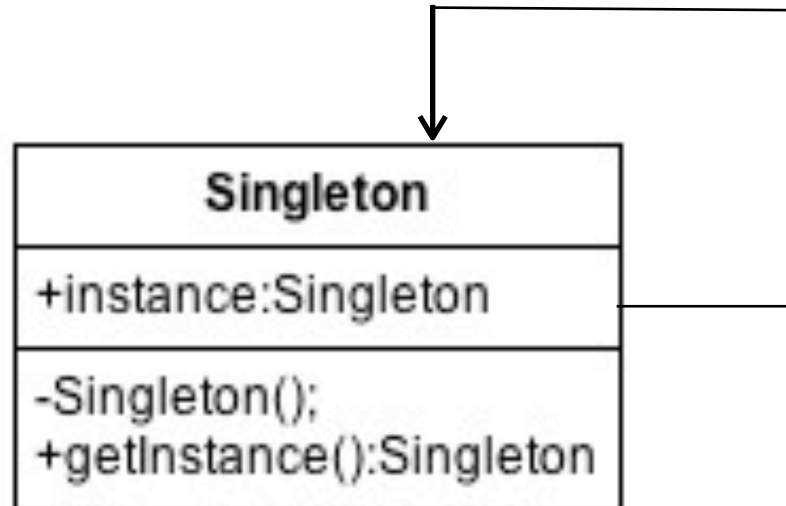
Human Computer Interaction Research
University of Nevada, Reno

Examples

- ★ Logging
- ★ Drivers
- ★ Caching
- ★ Security

- ★ incorrect behavior
- ★ overuse of resources
- ★ inconsistent results

class diagram

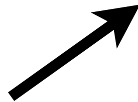


Solution

```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

   public static Singleton getInstance() {
       if (uniqueInstance == null) {
          uniqueInstance = new Singleton();
       }
       return uniqueInstance;
   }
}
```

How to instantiate object?

```
Singleton var = Singleton.getInstance()
```



class name not a variable

Problem: Not Thread Safe

```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

   public static Singleton getInstance() {
       if (uniqueInstance == null) {
          uniqueInstance = new Singleton();
       }
       return uniqueInstance;
   }
}
```

Problem: Not Thread Safe

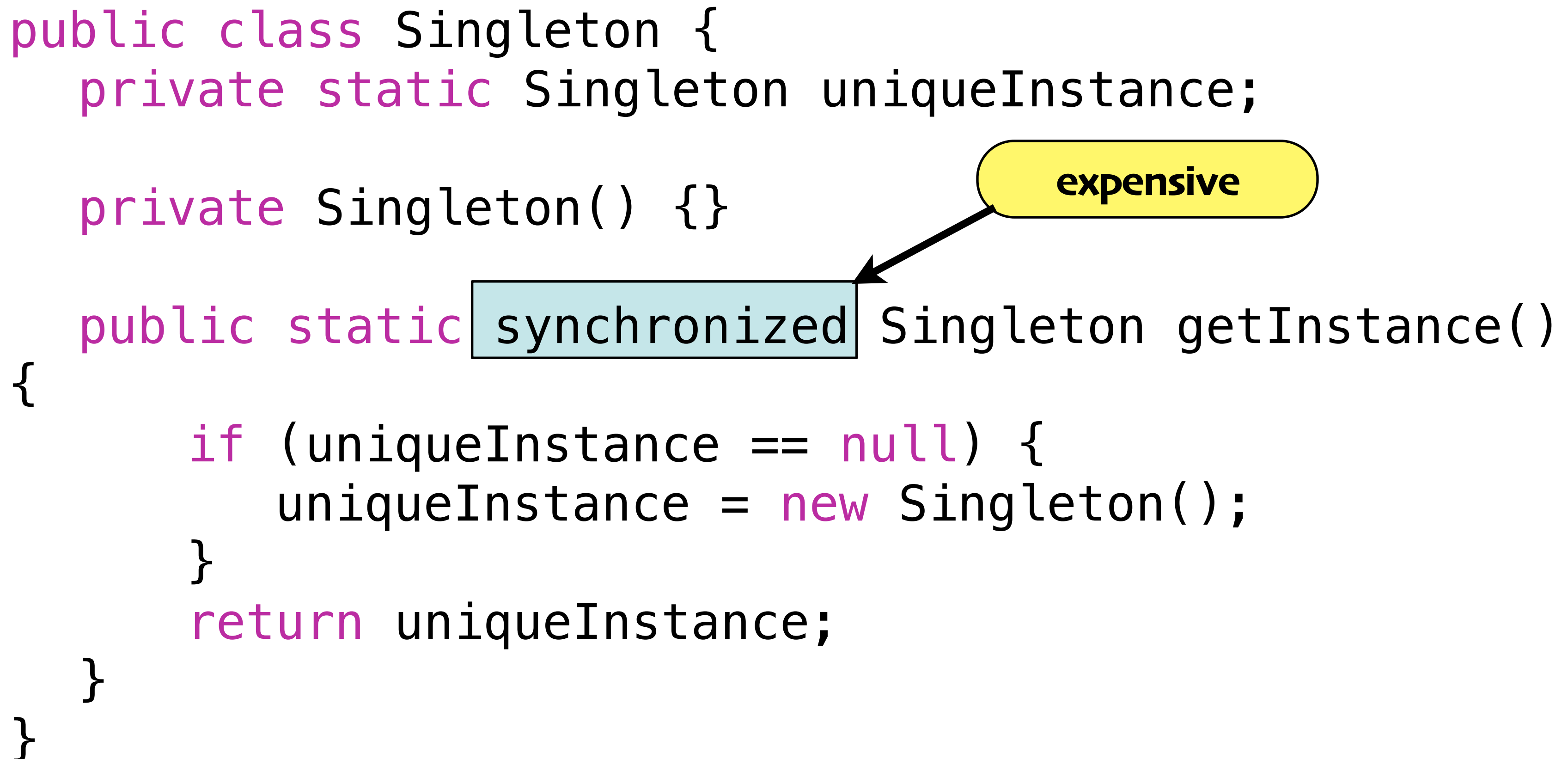
```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         uniqueInstance = new Singleton();
      }
      return uniqueInstance;
   }
}
```

Problem: Not Thread Safe

```java
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

   public static Singleton getInstance() {
       if (uniqueInstance == null) {
           uniqueInstance = new Singleton();
       }
       return uniqueInstance;
   }
}
```

Thread 2

solution 1: add synchronized




```
public class Singleton {
  private static Singleton uniqueInstance;

  private Singleton() {}

  public static synchronized Singleton getInstance()
{
      if (uniqueInstance == null) {
         uniqueInstance = new Singleton();
      }
      return uniqueInstance;
  }
}
```

solution 2: create eager instance

```
public class Singleton {
   private static Singleton uniqueInstance =
                                 new Singleton()

   private Singleton() {}

   public static synchronized Singleton getInstance()
{
       return uniqueInstance;
   }
}
```

JVM handles instance creation before threads have access

solution 2: double checked locking

```
public class Singleton {
   private volatile static Singleton uniqueInstance;

   private Singleton() {}

   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         synchronized (Singleton.class) {
            if (uniqueInstance == null) {
               uniqueInstance = new Singleton();
            }
         }
      }
      return uniqueInstance;
   }
}
```

handles var correctly when multiple threads init

only sync once

doesn't work in Java < 1.4

volatile

- The value of this variable will never be cached thread-locally
- Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself

Benefits Singleton

- ★ Ensures you only at most have one instance
- ★ Provides global access point
- ★ Examine your performance requirements to determine which implementation of the singleton works best.
- ★ Some implementations don't work with older version of Java (see book)

Exercise I

Logging

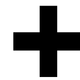
main app

```
public class LoggerTester {
    public static void main(String[] args) {
        Test test = new Test();
        System.out.println("Testing my logger");
        Logger.getInstance().log("testing one two three");
        test.run();
    }
}

class Test {
    public Test() {
    }
    void run() {
        Logger.getInstance().log("testing from within Test");
    }
}
```

Logger class

```
public class Logger {
    private static Logger uniqueInstance = new Logger();
    // eagerly created instance
    private int loggedmessages;

    private Logger() {
        loggedmessages=0;
    }
    public static Logger getInstance() {
        return uniqueInstance;
    }
    public void log( String msg )
    {
     loggedmessages++;
     System.out.println("LOG:"+ msg);
    }
}
```

Mixing patterns



Singleton

Factory Method

Human Computer Interaction Research
University of Nevada, Reno

Exercise II

A chocolate factory

Human Computer Interaction Research
University of Nevada, Reno

thread example

```
public class Something extends Thread{
    public void run()    {
        try {
                sleep(100);
            } catch (InterruptedException e) {}
        // do something
        }
```

```
Something t1 = new Something();
t1.start();
t2.stop();
```

Human Computer Interaction Research
University of Nevada, Reno

Main app

```
public class SingletonfactoryTest {
    public static void main(String[] args) {
        BarProducer t1 = new BarProducer(1);
        BarProducer t2 = new BarProducer(2);
        t1.start();
        t2.start();
    }
}
```

Thread

```
public class BarProducer extends Thread{
     Factory wonka;
     public int count=0; // each producer creates 50 bars
     public int id;

     public BarProducer(int identifier) {
          id=identifier;
          System.out.println("creating new Bar Producer with ID:"+id);
     }
     public void run()
        {
          while (count<50) {
               try {
                    sleep(100);
               } catch (InterruptedException e) {}
               count++;
               WonkaBarFactory.getInstance().create();
          }
       }}
```

Factory & Bar

```
public interface Factory {
    Bar create();
}
public abstract class Bar {
    public int id;
}

public class WonkaBar extends Bar {

    public WonkaBar(int identifier) {
        id = identifier;
    }
}
```

Concrete Singleton Factory

```
public class WonkaBarFactory implements Factory {
     private int counter=0;
     private static WonkaBarFactory uniqueInstance; // = new WonkaBarFactory();

     private WonkaBarFactory() { }

     public Bar create() {
          Bar bar = new WonkaBar(counter++);
          System.out.println("new Wonka bar created with id:" + counter);
          return bar;
     }

public static synchronized WonkaBarFactory getInstance() {
          if (uniqueInstance==null) {
               uniqueInstance= new WonkaBarFactory();
          }
          return uniqueInstance;
         }
}
```

Concrete Singleton Factory

```
public class WonkaBarFactory implements Factory {
    private int counter=0;
    private static WonkaBarFactory uniqueInstance; // = new WonkaBarFactory();

    private WonkaBarFactory() { }

    public synchronized Bar create() {
        Bar bar = new WonkaBar(counter++);
        System.out.println("new Wonka bar created with id:" + counter);
        return bar;
    }

public static synchronized WonkaBarFactory getInstance() {
        if (uniqueInstance==null) {
            uniqueInstance= new WonkaBarFactory();
        }
        return uniqueInstance;
      }
}
```